Specifying Optimizations in Attribute Grammars Using Temporal Logic

Mathias Staricka

Submitted under the supervision of Dr. Eric Van Wyk to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Computer Science.

May 9, 2014

Abstract

Modern compilers perform many optimizations to code in order to increase its execution speed and efficiency. These optimizations generally involve a complex analysis of the program being compiled in order to determine where code can be optimized. The algorithms to perform these analyses are complex and can be difficult both to understand and to implement. In order to make the specification of these analyses simpler and easier to understand, we have developed a system for specifying optimizations using temporal logic.

To perform these optimizations, we first augment a language's attribute grammar specification to generate a control flow graph which has been annotated with predicates giving details necessary for the analysis. This control flow graph along with a temporal logic formula is provided as input to a model checker which determines which points in the program's control flow graph correspond with program statements that can be optimized. Finally, these statements are transformed accordingly to generate a new, optimized abstract syntax tree which can be used for code generation.

Contents

Chapter 1

Introduction

In order to ensure fast code, many compilers perform optimizations to programs to speed up the execution of the program. These optimizations often involve sophisticated analyses of the program to determine their applicability. These analyses are complicated enough, that it is easy to make mistakes during their implementation and when looking at the implementation it is difficult to determine what the analysis is actually doing.

In this thesis, we seek to demonstrate a system for simplifying the specification of these optimizations through the use of computation tree logic (CTL). Previous work has used CTL and pattern matching for the specification of optimizations [3] and for proving the correctness of these optimizations [4], [5]. In particular, we build on the work in [6] by extending an attribute grammar language specification system with a framework for checking CTL formulas over a program's control flow graph.

As an example, take dead code elimination. Dead code elimination removes assignments to variables that have no effect on the program's execution. For example take the program:

```
a := 0;
b := 0;
read(c);
while(c){
        read(a);
        b = b + a;
        c = c - 1;
}
print(b);
```

In this program the initial assignment to a is dead since in any execution path through the program, the variable a is redefined (by the read statement) before ever being used. Thus the program can be optimized to:

```
b := 0;
read(c);
while(c){
        read(a);
        b = b + a;
        c = c - 1;
}
print(b);
```

This analysis can be formalized as an analysis over the program's control flow graph. A control flow graph is a directed graph modeling a program. Each node represents a program statement or condition. Edges represent paths execution can take through the program. For example, an if-then-else statement would be represented as a node for the condition with an edge going to the then branch and an edge going to the else branch.

The control flow graph of this example program is shown in Figure 1.1. This control flow graph is annotated with the properties `defs` and `uses`. Nodes are annotated by $defs(x)$ to indicate that the variable x is defined at that node. Nodes are annotated by $uses(x)$ to indicate that the variable x is used at that node. These properties provide information necessary to perform the liveness analysis used to determine whether assignments can be eliminated.

We say that a node corresponding to an assignment is dead if on all paths

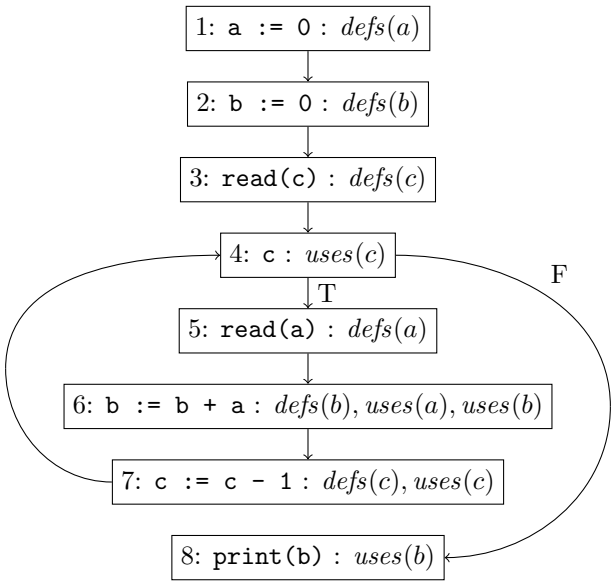

Figure 1.1: Control flow graph of the program to be optimized

starting at that node, the variable is not used again until either it is redefined or the program terminates. This condition can be expressed as the CTL formula

$$AX\ A[\neg uses(x) \wedge \neg defs(x)\ W\ defs(x) \wedge \neg uses(x)]$$

which states that for all succeeding nodes, on all paths beginning at those nodes, the variable x is not used and not defined until the variable x is defined and not used in that definition.

This formula provides a formal specification of the analysis which can be provided along with the control flow graph to a tool called a model checker to determine whether the formula holds at any node in the graph. We have built such a model checker as part of our project. When run on this formula and the CFG in 1.1, our model checker produces the result:

$$(1, [(x, a)], [])$$

which indicates that node 1 of the control flow graph satisfies this formula where x is the program variable a. Therefore the statement at node 1 is a candidate for removal.

We will present a framework in which the abstract syntax tree generated by an attribute grammar system is used to generate the program's control flow graph annotated with properties. These properties are then used by a model checker to identify nodes in the graph which satisfy the formula. Finally, the results from the model checker are used to transform the abstract syntax tree, generating an optimized version of the program.

In Chapter 2, we discuss the previous work upon which our system is built. Section 2.1 discusses the details of using an attribute grammar system to generate a control flow graph annotated with properties. Section 2.2 discusses in depth the syntax and semantics of CTL-FV, the extension of computation tree logic which our model checker uses.

In Chapter 3, we discuss the model checker we built for our system. Section 3.1 details the representation of infinite sets which the model checker utilizes to perform checking over arbitrary properties with potentially infinite domains. Section 3.2 discusses the algorithms used by the model checker to perform operations on these sets.

In Chapter 4, we provide several examples of the application of the model checker to perform diverse compiler optimizations. Section 4.1 explains how optimizations are applied to the abstract syntax tree. Section 4.2 provides a detailed example of dead code elimination. Section 4.3 provides an example of constant propogation. Section 4.4 provides an example of code motion through loop invariant hoisting.

In Chapter 5, we describe next steps to build upon this work.

Chapter 2

Background

2.1 CFG with Properties

The first step of developing this system is to derive the model over which the CTL formulas will be applied. Since most optimizations use a data flow analysis, it is natural to use the control flow graph of the program. For the purposes of this thesis we will assume a structured programming language (i.e. a language without gotos) is being used. Using an attribute grammar system, it is fairly straight-forward to generate the control flow graph from the program's abstract syntax tree.

Each statement has a synthesized attribute called `entry` which is a pointer to the node of the control flow graph where the statement begins (in some cases such as short-circuit evaluation of boolean expressions, a statement may encompass multiple nodes in the control flow graph). Each statement also has an inherited attribute called `succ` which is a pointer to the control flow graph node of the next statement.

Two constructors are used for creating the control flow graph nodes. The

```
aspect production sequence
s::Stmt ::= s1::Stmt s2::Stmt
{
    s.entry = s1.entry;
    s1.succ = s2.entry;
    s2.succ = s.succ;
}

aspect production while
s::Stmt ::= cond::Expr body::Stmt
{
    s.entry = branchNode(body.entry, s.succ, cond.uses);
    body.succ = s.entry;
}

aspect production assignment
s::Stmt ::= id::ID e::Expr
{
    s.entry = atomicNode(s.succ,
        [pair("defs", id.lexeme)] ++ e.uses);
}
```

Figure 2.1: Examples of attribute grammar code used to construct control flow graphs for sequence, while, and assignment statements

`atomicNode` constructor takes a single successor and a list of predicates which are true at the node. The `branchNode` constructor takes two successors, one for the true branch and one for the false branch, and a list of predicates which are true at the node. These predicates are determined by the abstract syntax tree as the control flow graph is being generated. Figure 2.1 shows some examples of the construction of the control flow graph.

The predicates with which the control flow graph is annotated take the form of some property parametrized with a value. For example, in dead code elimination, `defs` and `uses` properties are used. A node which assigns a value to the variable x is annotated with the predicate $defs(x)$. A node which uses

the value of the variable x is annotated with the predicate $uses(x)$. Figure 1.1 shows a simple control flow graph annotated with `defs` and `uses` predicates.

2.2 CTL-FV

Having generated the structure over which to perform the analysis, the next piece is the logic which formalizes the analysis. For our system we use a variant of computation tree logic (CTL) which has been extended with free variables. CTL is a temporal logic which adds temporal operators to sentential logic allowing reasoning to be performed over a notion of time (control flow between statements in our case).

2.2.1 Syntax

CTL formulas are formed similarly to sentential logic formulas but with the addition of temporal operators. Each temporal operator is made up of two letters, a quantifier A, $\overleftarrow{A}$, E, or $\overleftarrow{E}$ paired with either X, F, G, U, or W. Operators containing X, F, or G are unary and operators containing U or W are binary. In the extension of CTL we are using, simple sentential statements are replaced by predicates of the form $p(x)$ where p is a property and x is a free variable. The grammar for CTL-FV is:

$$\phi ::= p(x)|\neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \rightarrow \phi|\phi \leftrightarrow \phi|A\ \Phi|\overleftarrow{A}\ \Phi|E\ \Phi|\overleftarrow{E}\ \Phi$$

$$\Phi ::= X\ \phi|F\ \phi|G\ \phi|[\phi\ U\ \phi]|[\phi\ W\ \phi]$$

The semantics of these formulas will be covered in the following subsections.

Satisfaction Notation	Meaning
$n \vDash_f \neg\phi$	not $(n \vDash_f \phi)$
$n \vDash_f \phi_1 \wedge \phi_2$	$n \vDash_f \phi_1$ and $n \vDash_f \phi_2$
$n \vDash_f \phi_1 \vee \phi_2$	$n \vDash_f \phi_1$ or $n \vDash_f \phi_2$
$n \vDash_f \phi_1 \rightarrow \phi_2$	$n \vDash_f \phi_1 \implies n \vDash_f \phi_2$
$n \vDash_f \phi_1 \leftrightarrow \phi_2$	$n \vDash_f \phi_1 \iff n \vDash_f \phi_2$

Table 2.1: The definition of satisfaction for nontemporal operators

2.2.2 Free Variable Instantiation

In CTL-FV, the sentential statements used as atomic formulas in standard CTL are replaced by predicates parametrized by free variables. These predicates take the form of a property (e.g. *defs* and *uses*) and a free variable. Each property p has an associated domain D_p of values which associated free variables can take. For example, in the case of the *defs* predicates used in dead code elimination, $D_{defs} = \{\text{program variables occurring in the program being analyzed}\}$.

Definition 1. Let ϕ be a CTL-FV formula using properties $p_1, p_2, \ldots p_n$ and whose set of free variables is V. An *instantiation* for the formula ϕ, is a function $f : V \rightarrow D_{p_1} \cup D_{p_2} \cup \ldots \cup D_{p_n}$ such that $f(x) \in D_{p_i}$ for each property p_i for which x is used as a parameter.

Definition 2. Let f be an instantiation for the atomic formula $p(x)$ and n be a node. $n \vDash_f p(x)$ if the node n is annotated with the predicate $p(f(x))$. It is said that f satisfies $p(x)$ at the node n.

Table 2.1 gives the definition of satisfaction for formulas containing nontemporal operators. Satisfaction for temporal operators are defined in the next section.

2.2.3 Temporal Operators

The temporal operators allow formulas to express properties over paths in the program's control flow graph. Temporal operators beginning with A state that a given property holds over **all** paths beginning at a given node. Temporal operators beginning with E state that a given property holds over some path (there **exists** a path) beginning at a given node. Operators beginning with $\overleftarrow{A}$ or $\overleftarrow{E}$ behave similarly to their counterparts without the arrow, but express properties over paths backwards (the reverse of the order of execution) in the control flow graph. Table 2.2 gives the definition of each temporal operator.

2.2.4 Sets of Satisfying Instantiations

In practice, a model checker would not check every possible instantiation for a formula individually. Instead, a systematic method of building up the set of all instantiations which satisfy a formula at each node in the control flow graph is needed. To this end, we define the set $Sat(\phi, N)$.

Definition 3. Let F_ϕ be the set of all instantiations for ϕ. $Sat(\phi, N) = \{(n, f) \in N \times F_\phi : n \vDash_f \phi\}$. $Sat(\phi, N)$ is the set of pairs (n, f) of nodes in N and instantiations such that the instantiation f satisfies the formula ϕ at the node n.

For an atomic formula $p(x)$, it is simple to construct:

$$Sat(p(x), N) = \{(n, f) \in N \times F_{p(x)} : \text{the node } n \text{ is annotated with the predicate } p(f(x))\}.$$

Given this starting point, formulas ψ formed by joining atomic formulas with nontemporal operators can have their corresponding sets $Sat(\psi, N)$ constructed by performing set operations on the sets corresponding with the atomic formulas.

It follows from Definitions 2 and 3 that:

- $Sat(\phi_1 \wedge \phi_2, N) = Sat(\phi_1, N) \cap Sat(\phi_2, N)$,
- $Sat(\phi_1 \vee \phi_2, N) = Sat(\phi_1, N) \cup Sat(\phi_2, N)$, and
- $Sat(\neg\phi, N) = (N \times F_\phi) \setminus Sat(\phi, N)$.

Temporal operators are more involved, as they reason over paths through the graph. It is still possible, however, to derive their sets of satisfying instantiations from those of the subformulas. For example, $Sat(AX\ \phi, N) = \{(n, f) : \forall m \in N\ [m \text{ is a successor of n } \implies (m, f) \in Sat(\phi, N)]\}$.

Operator	Name	Definition
$n_0 \vDash_f AX\phi$	all next	on all paths $n_0, n_1, \ldots$ beginning at node n_0: $n_1 \vDash_f \phi$
$n_0 \vDash_f EX\phi$	exists next	there exists a path $n_0, n_1, \ldots$ beginning at node n_0 such that $n_1 \vDash_f \phi$
$n_0 \vDash_f AF\phi$	all eventually	on all paths $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0 there is a node n_i along the path such that $n_i \vDash_f \phi$
$n_0 \vDash_f EF\phi$	exists eventually	there exists a path $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0 such that there is a node n_i along the path with $n_i \vDash_f \phi$
$n_0 \vDash_f AG\phi$	all globally	on all paths $n_0, n_1, \ldots$ beginning at node n_0, $n_i \vDash_f \phi$ holds for every node n_i along the path
$n_0 \vDash_f EG\phi$	exists globally	there exists a path $n_0, n_i, \ldots$ beginning at node n_0, such that $n_i \vDash_f \phi$ for every node n_i along the path
$n_0 \vDash_f A[\phi_1 \; U \; \phi_2]$	all until	on each path $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0, there exists a node n_i such that $n_i \vDash_f \phi_2$ and for every node n_j occurring earlier in the path $n_j \vDash_f \phi_1$
$n_0 \vDash_f E[\phi_1 \; U \; \phi_2]$	exists until	there exists a path $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0 and a node n_i along that path such that $n_i \vDash_f \phi_2$ and for every node n_j occurring earlier in the path $n_j \vDash_f \phi_1$
$n_0 \vDash_f A[\phi_1 \; W \; \phi_2]$	all weak until	on each path $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0, either $n_i \vDash_f \phi_1$ for every node n_i along the path, or there exists a node n_i on the path such that $n_i \vDash_f \phi_2$ and for every node n_j occurring earlier in the path $n_j \vDash_f \phi_1$
$n_0 \vDash_f E[\phi_1 \; W \; \phi_2]$	exists weak until	there exists a path $n_0, n_1, \ldots, n_i, \ldots$ beginning at node n_0 such that either $n_i \vDash_f \phi_1$ for every node n_i along the path, or there exists a node n_i on the path such that $n_i \vDash_f \phi_2$ and for every node n_j occurring earlier in the path $n_j \vDash_f \phi_1$

Table 2.2: The definition of each temporal operator. Operators with $\overleftarrow{A}$ or $\overleftarrow{E}$ are defined the same way but over paths in the reverse direction (i.e paths from nodes to their predecessors as opposed to their successors).

Chapter 3

CTL-FV Model Checker

Having established a method for generating a program's control flow graph and CTL-FV as a language for reasoning over that graph, the next step is a system for checking CTL-FV formulas over that graph. This is accomplished with a program called a model checker which analyzes the graph to determine which instantiations will satisfy the formula at each node.

The result returned by the model checker given a formula ϕ and a control flow graph with nodes N is the set $Sat(\phi, N)$. As established in 2.2.4, these sets can be constructed by initially determining $Sat(\psi, N)$ for each atomic subformula of ψ and performing set operations on these sets. The complication comes in how to represent these sets.

3.1 Representation of the Result

One way to think about these sets is as a specification for instantiations giving restrictions on what values the instantiations can assign to each free variable at each node. For example, consider the control flow graph in Figure 1.1. Let N be the set of nodes in this graph. The set $Sat(uses(x), N)$ can be thought of as

follows:

n	restrictions on f
4	$f(x) = c$
6	$f(x) = a$ or $f(x) = b$
7	$f(x) = c$
8	$f(x) = b$

Each of these restrictions is of the form $f(x) = a$ making it simple to represent them as pairs of free variables and values (i.e. $f(x) = a$ is represented as (x, a)). Intersections can then be performed by keeping a list of all restrictions occurring at each node. Unions can be performed by keeping multiple lists (one for each possibility). For example, using the same graph as above, $Sat(\mathit{defs}(x) \wedge \mathit{uses}(y))$ can be represented as follows:

n	restrictions on f	representation of restrictions
6	$f(x) = b$ and $[f(y) = a$ or $f(y) = b)]$	$[(x, b), (y, a)]$
		$[(x, b), (y, b)]$
7	$f(x) = c$ and $f(y) = c$	$[(x, c), (y, c)]$

There is a complication when formulas containing $\neg$ are considered. Then instead of having restrictions of the form $f(x) = a$, we have restrictions of the form $f(x) \neq a$. To keep the same representation for restrictions would require exploding the number of lists of restrictions, producing one for each value of in the property's domain other than a. For example $Sat(\neg \mathit{uses}(x), N)$ would be

represented as follows:

n	restrictions on f	representation of restrictions
1	$f(x) = a$ or $f(x) = b$ or $f(x) = c$	$[(x, a)]$ $[(x, b)]$ $[(x, c)]$
$\vdots$	$\vdots$	$\vdots$
4	$f(x) = a$ or $f(x) = b$	$[(x, a)]$ $[(x, b)]$
$\vdots$	$\vdots$	$\vdots$

In order to avoid this explosion in the size of our representation and to accommodate properties with potentially infinite domains, we introduce a second class of pairs corresponding to restrictions of the form $f(x) \neq a$. Then the representation for $Sat(\neg uses(x), N)$ becomes:

		representation of restrictions	
n	restrictions on f	$f(x) = a$	$f(x) \neq a$
1	none	[]	[]
2	none	[]	[]
3	none	[]	[]
4	$f(x) \neq c$	[]	$[(x, c)]$
5	none	[]	[]
6	$f(x) \neq a$ and $f(x) \neq b$	[]	$[(x, a), (x, b)]$
7	$f(x) \neq c$	[]	$[(x, c)]$
8	$f(x) \neq b$	[]	$[(x, b)]$

Finally, each row of the table can be represented as a triplet (n, γ, δ) where γ is the first class of restrictions and δ is the second class of restrictions. So, for

example the set $Sat(uses(x) \wedge \neg defs(y))$ is represented as the list of triples:

$$(4, [(x, c)], []),$$
$$(6, [(x, a)], [(y, b)]),$$
$$(6, [(x, b)], [(y, b)]),$$
$$(7, [(x, c)], [(y, c)]),$$
$$(8, [(x, b)], [])$$

3.2 Set Operations of this Representation

When performing analysis of CTL-FV formulas, the model checker needs to perform several common set operations. As discussed in 2.2.4, nontemporal operators map directly to the standard set operations union, intersection, and complement. While temporal operators do not map nicely to this operations, the computation required for the temporal operators does generally rely on unions and intersections as part of the computation.

3.2.1 Union

The representation used by the model checker makes performing unions very straight-forward. Given a list of triples α representing set A and a list of triples β representing set B, the representation of $A \cup B$ is simply the list of triples acquired by appending list β to list α.

3.2.2 Intersection

The intersection of two sets represented by lists α and β is computed by comparing the triples from the two lists pairwise. Each pair of one triple (m, γ_1, δ_1) from list α and one triple (n, γ_2, δ_2) from list β is compared to determine if

the intersection of the sets of instantiations represented by the two triples is nonempty. If this intersection is nonempty a triple is generated to be added to the representation of the intersection of the sets represented by α and β. To accomplish this, for each pair of triples (m, γ_1, δ_1) from α and (n, γ_2, δ_2) from β the following is performed:

1. If $m \neq n$, the intersection of the two triples is empty. Move on to the next pair of triples.

2. For each pair (x, a) in the list γ_1:

   (a) If there exists a pair (x, b) in the list γ_2 with $a \neq b$, the intersection of the two triples is empty. Move on to the next pair of triples.

   (b) If no such pair (x, b) exists, add (x, a) to a new list γ_3.

3. For each pair (x, a) in the list γ_2, if the pair (x, a) is not already in γ_3, add the pair (x, a) to γ_3.

4. For each pair (x, a) in the list δ_1:

   (a) If the pair (x, a) is in the list γ_3, the intersection of the two triples is empty. Move on to the next pair of triples.

   (b) Otherwise, if there does not exist a pair (x, b) in γ_3 with $a \neq b$, add the pair (x, a) to a new list δ_3.

5. For each pair (x, a) in the list δ_2:

   (a) If the pair (x, a) is in the list γ_3, the intersection of the two triples is empty. Move on to the next pair of triples.

   (b) Otherwise, if there does not exist a pair (x, b) in γ_3 with $a \neq b$ and if the pair (x, a) is not already in the list δ_3, add the pair (x, a) to δ_3.

6. If the pair of triples has not already been rejected in step 1, 2a, 4a, or 5a, the triple (m, γ_3, δ_3) represents the intersection of the pair of triples. Add this new triple to the representation of the intersection of α and β and move on to the next pair of triples.

The intersection of the sets represented by α and β is the set represented by the list made up of any triples generated by the above process.

3.2.3 Complement

As noted in 2.2.4, the complement operation is used when $\neg$ occurs in a formula. The model checker simplifies this calculation by transforming CTL-FV formulas to have any occurrences of $\neg$ occur only at atomic formulas. This transformation can be performed using De Morgan's laws and identities found in [1] and [2]. Table 3.1 shows some examples of these identities.

Not Propagation Identities
$n \vDash_f \neg(\phi \wedge \psi) \iff n \vDash_f \neg\phi \vee \neg\psi$
$n \vDash_f \neg(\phi \vee \psi) \iff n \vDash_f \neg\phi \wedge \neg\psi$
$n \vDash_f \neg AX\ \phi \iff n \vDash_f EX\ (\neg\phi)$
$n \vDash_f \neg EF\ \phi \iff n \vDash_f AG\ (\neg\phi)$
$n \vDash_f \neg EG\ \phi \iff n \vDash_f AF\ (\neg\phi)$
$n \vDash_f \neg A[\phi\ U\ \psi] \iff n \vDash_f E[\neg\psi\ U\ (\neg\phi \wedge \neg\psi)] \vee EG\ (\neg\psi)$
$n \vDash_f \neg A[\phi\ W\ \psi] \iff n \vDash_f E[\neg\psi\ U\ (\neg\phi \wedge \neg\psi)]$

Table 3.1: Examples of identities for propagating $\neg$ to atomic formulas

Thus we need only handle the special case of $\neg p(x)$. We can directly compute $Sat(\neg p(x), N)$ which is represented by the list of triples $(n, [], [(x, a)])$ where the node n is annotated with the predicate $p(a)$ (one triple occurs for each predicate for p occurring on node n) and $(n, [], [])$ where the node n is annotated with no predicates for the property p.

Chapter 4

Performing Optimizations

In this chapter, we will demonstrate the application of the CTL-FV model checker to perform optimizations on a program's abstract syntax tree. We first describe how optimizations are performed through transformation of the abstract syntax tree. We then provide examples of optimizations that can be performed with this system. The first example is the dead code elimination example we have been using throughout the thesis. Then, we will explain constant propagation which will demonstrate an analysis that is performed in the reverse direction (relative to program execution). Finally, we will cover loop invariant hoisting, a more complicated optimization which reasons about two nodes simultaneously in order to move a statement within a program.

4.1 Transforming the Abstract Syntax Tree

The current system we are using for applying optimizations performs transformations on the program's abstract syntax tree. When an optimization is performed, the attribute grammar system first generates the program's abstract syntax tree from which the program's control flow graph is derived. Next, the

model checker is run on the control flow graph with the desired optimization's CTL-FV formula. The list of instantiations generated by the model checker is propagated throughout the abstract syntax tree through an inherited attribute.

Given the instantiations generated by the model checker, each node in the abstract syntax tree inspects the instantiations and determines whether it is a candidate for a transformation. Each node in the abstract syntax tree has a synthesized attribute called `optimized` which gives a transformed version of that node based on the optimization being performed. If the node is not a candidate for transformation (either the optimization does not perform transformations on that type of node or there were no suitable instantiations for that node found by the model checker), the node sets its `optimized` attribute to a the same type of node with its children set to the `optimized` attribute on the original children. If the node is a candidate for transformation, it will set its `optimized` attribute to the transformed version of the node.

If multiple optimizations (or multiple passes of the same optimization) are to be performed, these optimizations are performed one at a time. After each pass, a new control flow graph is generated from the transformed abstract syntax tree. The next optimization is then performed on the new control flow graph. This results in a back-and-forth between the two representations of the program. In future, it may be desirable to modify the system to perform the transformations directly on the control flow graph in order to eliminate the overhead involved in moving between the two representations.

4.2 Dead Code Elimination

As described in the Chapter 1, dead code elimination seeks to eliminate assignments that go unused in the program. While programmers may not frequently include dead assignments in their programs, other optimizations, such as the

constant propagation optimization described in Section 4.3, may result in assignments that previously were not dead being dead in the transformed program. Thus dead code elimination can serve as a clean-up optimization after other optimizations are applied.

4.2.1 Properties

Dead code elimination uses two properties, both which have been used as examples throughout the thesis.

Definition 4. The *defs* property specifies which program variables are defined at a node. $D_{defs} = \{a : a \text{ is a program variable in the program being analyzed}\}$. A node is annotated with the predicate $defs(a)$ if the program variable a is defined at that node (such as with an assignment or read statement).

Definition 5. The *uses* property specifies which program variables are used at a node. $D_{uses} = \{a : a \text{ is a program variable in the program being analyzed}\}$. A node is annotated with the predicate $uses(a)$ if the value of the program variable a is used at that node.

4.2.2 CTL-FV Formula

The CTL-FV formula used for dead code elimination is as follows:

$$defs(x) \wedge AX\ A[\neg uses(x) \wedge \neg defs(x)\ W\ defs(x) \wedge \neg uses(x)].$$

An assignment is dead if along all paths beginning at the assignment's successors, the variable defined by that assignment is not used again before either the end of the program is reached or the variable is redefined (and its value is not used in that definition).

Note that the formula presented here is different from the formula presented in the introduction. This version is prefixed with $defs(x)\wedge$. While not strictly necessary (and left out for simplicity in the introduction), this additional predicate serves to simplify the results from the model checker. The predicate ensures that x can only be instantiated to the variable defined at the node being checked. Otherwise the formula would be satisfied by all program variables which are dead at the node being checked, even those which are not defined at the node being checked and thus irrelevant to dead code elimination. This allows the optimization to be applied to any assignment statement whose node is in the list of results without requiring the restrictions on the instantiations to be checked.

4.2.3 Transformation

An assignment statement `a := e` at node n is transformed to `skip` if there exists a triple in the model checker's results which begins with n. A skip statement is a no-op which can be left out during code generation. All other nodes are left as is.

4.2.4 Example Program

Consider the program in Figure 4.1 and the corresponding control flow graph in Figure 4.2. Consider the assignment at node 1. Along the path $2, 3, 4, \ldots$, a is not used until it is defined in node 4. Along the path $2, 3, 6, 7, 11$ a is not used until the end of the program is reached (thus the "weak" part of the until comes into play). Along the path $2, 3, 6, 7, 8, \ldots$ the variable a is not used until it is defined in node 8. Thus the CTL-FV formula for dead code elimination is satisfied at node 1 by the instantiation $f(x) = a$. Thus the model checker

```
a := 0;
read(c);
if(c){
        read(a);
        print(a);
} else {
        read(b);
        while(b > 0){
                read(a);
                print(a);
                b := b - 1;
        }
}
print("done");
```

Figure 4.1: An example program for dead code elimination

produces the result

$$(1, [(x, a)], []),$$

and the assignment at node 1 is replaced by a skip.

4.3 Constant Propagation

Constant propagation is an optimization which replaces references to a variable which has been assigned a constant value with the constant itself. This serves to eliminate unnecessary memory accesses when the value of a variable is in fact known at compile time. Unlike dead code analysis which performs an analysis forwards in the control flow graph, constant propagation uses an analysis backwards in the control flow graph.

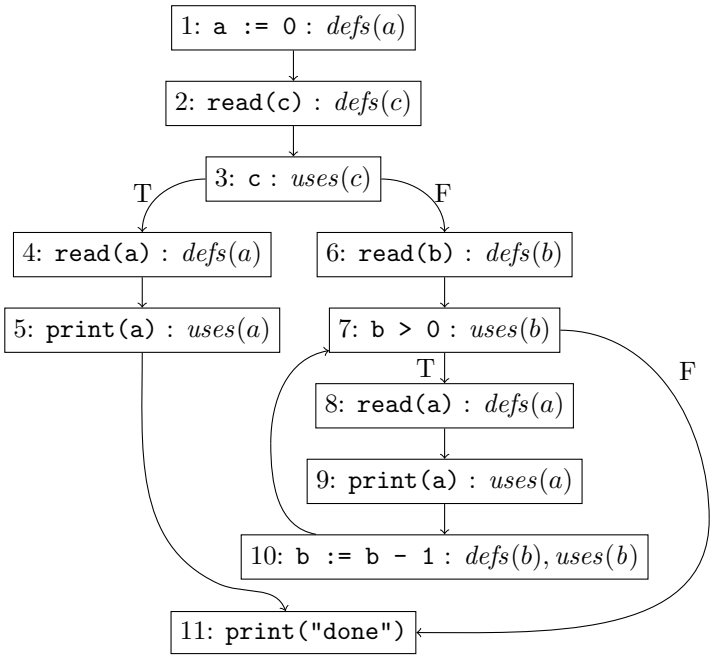

Figure 4.2: The control flow graph for the dead code elimination example program

4.3.1 Properties

In addition to the *defs* property defined in 4.2.1, constant propagation uses a new property called *const*.

Definition 6. The *const* property specifies that the statement associated with the node assigns a constant value to the variable it defines. $D_{const} = \{n : n$ is an integer literal used in the program being analyzed$\}$. A node is annotated with the $const(n)$ if the node is an assignment statement which assigns an integer literal to the variable (e.g. `a := 42` would be annotated with $const(42)$).

4.3.2 CTL-FV Formula

The CTL-FV formula used for constant propagation is as follows:

$$\overleftarrow{A} X \overleftarrow{A}[\neg defs(var) \; U \; (defs(var) \wedge const(val))]$$

A variable used in an expression at a given node can be replaced with an integer literal if on all paths backwards from the node, the last definition of that variable assigned the same integer literal to that variable.

4.3.3 Transformation

Every statement containing an expression (e.g. `a := e` and `if(e)`) transforms the expression as follows. In a statement at node n, for each triple $(n, [(var, a), (val, m)], [])$ in the model checker's results, each occurrence of the variable a in expressions at that node is replaced by the integer literal m. All other nodes are left as is.

4.3.4 Example Program

Consider the example program:

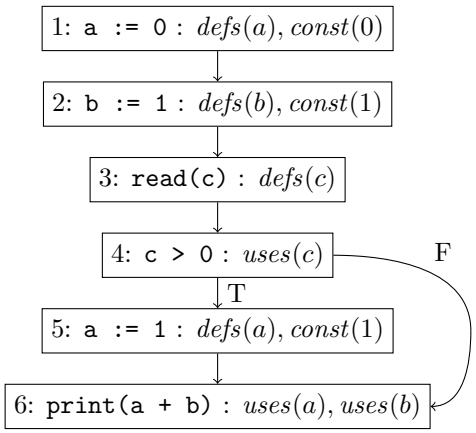

Figure 4.3: The control flow graph for the constant propagation example program

```
a := 0;
b := 1;
read(c);
if (c > 0) {
        a := 1;
}
print(a+b);
```

Figure 4.3 shows the control flow graph for the program.

This program has three constant assignments, one assigning a constant to a at node 1, one assigning a constant to b at node 2, and one assigning a constant to a at node 5. The only uses of a and b occur at node 6. First consider the use of a at node 6. Along the reverse path $4, 3, 2, 1$, the first definition encountered for a is the constant definition at node 1 which assigns 0 to a. However, along the path $5, 4, 3, 2, 1$, the first definition encountered for a is the definition at node 6 which assigns 1 to a. Since the two definitions assign different constants, no instantiation f with $f(var) = a$ exists which satisfies the formula at 6.

Now consider the use of b at node 6. Along the path $4, 3, 2, 1$, the first definition encountered for b is the definition at node 2. Along the path $5, 4, 3, 2, 1$, the first definition encountered for b is once again the definition at node 2. Thus

the instantiation f with $f(var) = b$ and $f(val) = 1$ satisfies the formula at node 6.

The results from the model checker on this example program are:

$$(2, [(var, a), (val, 0)], [])$$

$$(3, [(var, a), (val, 0)], [])$$

$$(3, [(var, b), (val, 1)], [])$$

$$(4, [(var, a), (val, 0)], [])$$

$$(4, [(var, b), (val, 1)], [])$$

$$(5, [(var, a), (val, 0)], [])$$

$$(5, [(var, b), (val, 1)], [])$$

$$(6, [(var, b), (val, 1)], [])$$

Since a and b are only used at node 6, the only pertinent result is $(6, [(var, b), (val, 1)], [])$ which we anticipated. Thus the statement at node 6 is changed to `print(a + 1)`.

4.4 Loop Invariant Hoisting

Loop invariant hoisting is an optimization which detects statements within loops that are loop invariants and moves those statements outside the loop. The version of loop invariant hoisting presented here utilizes the fact that optimizations are being performed on control flow graphs generated directly from the abstract syntax tree of structured programs. The CTL-FV formula for this optimization is made up of two subformulas. One which gives criteria that must be satisfied by the expression that is being moved and the other which gives criteria that must be satisfied by the destination to which the statement will be moved.

4.4.1 Properties

Loop invariant hoisting uses several new properties. The *trans* and *assignExpr* properties allow reasoning over the expressions in the program (as opposed to just variables). The *node*, *loopStart*, and *closetLoop* properties are all used for identifying locations within the control flow graph. In particular the *loopStart*, and *closetLoop* properties are used to select a unique destination for the statement being moved, but assume that the optimization is being performed on a structured program.

Definition 7. The *trans* property specifies which expressions are "transparent" at a node. $D_{trans} = \{e : e \text{ is a an expression in the program being analyzed}\}$. An expression e is considered transparent at a node n if none of the variables which occur in e are defined at n.

Definition 8. The *assignExpr* property specifies the expression being assigned to a variable at assignment nodes. $D_{assignExpr} = \{e : e \text{ is a an expression in the program being analyzed}\}$. A node is annotated with the predicate $assignExpr(e)$ if the node is an assignment statement with the expression e on the right side of the assignment (i.e. an assignment `a := e`).

Definition 9. The *node* property specifies the unique ID assigned to the node. $D_{node} = \{n : n \text{ is the ID of a node in the control flow graph}\}$. A node n will be annotated with the parameter $node(n)$.

Definition 10. The *loopStart* property specifies that a loop begins at the node. $D_{loopStart} = \{n : n \text{ is an ID assigned to a (structured) loop in the program}\}$. A node is annotated with the predicate $loopStart(n)$, where n is a unique ID assigned to the loop, if the node is the entry node of a for, while, or other loop construct.

Definition 11. The *closestLoop* property specifies the innermost loop in which

the statement occurs. $D_{closestLoop} = \{n : n$ is an ID assigned to a (structured) loop in the program$\}$. A node is annotated with the predicate *closestLoop*(n) if the statement occurs within a loop with the loop n being the inner-most loop in which the statement occurs.

For example, in the control flow graph in Figure 4.5, the node 3 which corresponds with the outer while loop is annotated with *loopStart*$(L1)$ and the node 4 which corresponds with the inner while loop is annotated with *loopStart*$(L2)$. The nodes 4 and 8 which occur within the outer loop but not the inner loop are annotated with *closestLoop*$(L1)$ and the nodes 5, 6, and 7 which occur within the inner loop are annotated with *closestLoop*$(L2)$.

4.4.2 CTL Formula

The CTL-FV formula used for loop invariant hoisting found here has been adapted from the formula found in [4] to be compatible with our model checking system, and is as follows:

$$node(p) \wedge loopStart(l) \wedge A[\neg uses(x)\ W\ node(q)]$$
$$\wedge EF(node(q) \wedge closestLoop(l) \wedge assignExpr(e) \wedge defs(x) \wedge (\neg uses(x)$$
$$\wedge \overleftarrow{A}[(\neg defs(x) \vee node(q)) \wedge trans(e)\ W\ \neg closestLoop(l) \wedge \neg loopStart(l)]))$$

An assignment statement `a := e` at node q is a candidate for loop invariant hoisting if their exists a destination node p such that:

1. on all paths starting at node p, the variable a is not used until either the node q or the end of the program is reached,

2. the node q does not use the variable a (i.e. the variable a does not occur in the expression e), and

3. on all paths backwards in the graph starting at node q, each node is either the node q or does not define the variable a and the expression e is transparent at each node until the destination or the beginning of the program is reached.

We add the additional constraint that the destination for the statement must be immediately outside the inner-most loop in which the statement occurs in order to ensure that only one destination is found.

In the CTL-FV formula seen above, the first subformula $node(p) \wedge loopStart(l) \wedge A[\neg uses(x)\ W\ node(q)]$ specifies what criteria must be satisfied by the destination node (as described in item 1 above). The remainder of the formula is wrapped in an EF operator, indicating that there exists some source node q found later in the graph which satisfies the criteria stated in items 2 and 3 above.

4.4.3 Transformation

For each triple $(n_0, [(p, n_0), (l, m), (x, a), (q, n_1), (e, f)], [])$ occurring in the model checker's results, the following transformation is performed. The node n_0 (which by definition of the *loopStart* predicate must be associated with a loop construct) has its abstract syntax tree node replaced by a sequence statement $\mathtt{S}_{\mathtt{n}_1}; \mathtt{S}_{\mathtt{n}_0}$ where $\mathtt{S}_{\mathtt{n}_1}$ is the statement originally occurring at node n_1 and $\mathtt{S}_{\mathtt{n}_0}$ is the statement originally occurring at node n_0. The statement occurring at node n_1 is replaced by a `skip` statement.

4.4.4 Example Program

Consider the example program in Figure 4.4. The control flow graph (without the *trans* predicates) of the program is shown in Figure 4.5. Table 4.1 shows which expressions are transparent at each node.

```
read(a);
read(b);
while(a > 0){
        while(b > 0){
                print(a, b);
                c := 2;
                b := b - c;
        }
        a := a - 1;
}
print("done");
```

Figure 4.4: Example program for loop invariant hoisting

The assignment `c := 2` at node 6 is a loop invariant. Consider paths backwards from node 6 to the exit of the loop. On the path $6, 5, 4, 7, 6, 5, 4$ the expression 2 is always transparent and the only definitions of c occur at node 6 so the portion of the formula within the EF operator is satisfied at node 6 with the instantiation f such that $f(q) = 6, f(l) = L2, f(e) = 2$, and $f(x) = c$.

Now we consider the destination. The predicate $loopStart(L2)$ occurs at node 4. Along the path $4, 5, 6$ the variable c is not used before node 6 is reached. Along the path $4, 8, 3, 4, 5, 6$, the variable c is not used before node 6 is reached. Along the path $4, 8, 3, 9$ the variable c is not used before the end of the program is reached. Thus we conclude that formula for loop invariant hoisting is satisfied at node 4 by the instantiation f such that $f(p) = 4, f(q) = 6, f(l) = L2, f(e) = 2$, and $f(x) = c$. Therefore loop invariant hoisting can be applied to the program by moving the statement at node 6 outside the loop starting at node 4.

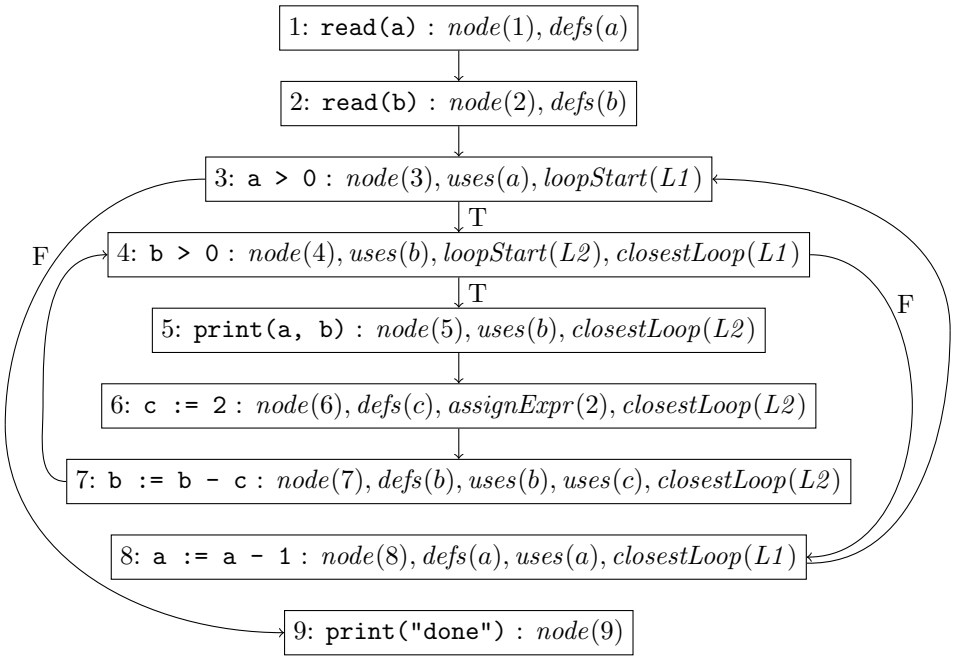

Figure 4.5: The control flow graph for the loop invariant hoisting example program with the *trans* predicates left off

Node	`a > 0`	`b > 0`	`a`	`b`	`2`	`b-c`	`a-1`
1		X		X	X	X	
2	X		X		X		X
3	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X
6	X	X	X	X	X		X
7	X		X		X		X
8		X		X	X	X	
9	X	X	X	X	X	X	X

Table 4.1: Each row indicates the expressions which are transparent at each node with an X. Each node is annotated with the predicate *trans*(e) for each expression e which is transparent at that node.

Chapter 5

Future Work

The system we have developed has several limitations. The program transformations performed by the optimizations are performed on the program's abstract syntax tree in the source language. This may in some cases be too high-level of a representation to perform all the optimizations desired. Further, when performing multiple passes of optimizations, a transformed abstract syntax tree is generated in each pass resulting in the control flow graph being regenerated in each pass. This overhead caused by moving back-and-forth between representations could be alleviated by modifying the system to perform transformations directly on the control flow graph which is being used for model checking.

The system we designed was primarily a proof-of-concept system. The model checker is not optimized and is likely too slow to be used on actual production code. To make the system more practical for use on actual code, it would be desirable to either improve the algorithms used by the model checker or possibly integrate the system with an already established model checker.

Finally, we built our system as an extension to a minimal language using an attribute grammar system called Silver. It would be desirable to separate

the system from that language's specification and make an extension to Silver itself. This would allow the easy application of the system to other languages specified in Silver. This would support further exploration of applications of the system to not only general optimizations, but also optimizations as part of domain-specific language extensions.

Chapter 6

Conclusion

Compiler optimizations are important tools for producing faster and more efficient code, however they involve complicated analyses which can be difficult both to implement and to understand. By using a temporal logic to specify the analysis used by these optimizations, the complicated implementation details of the analysis can be abstracted away allowing a more clearly understood specification.

In this thesis, we presented a framework including a CTL-FV model checker which allows for the simple specification of optimizations as part of an attribute grammar language specification. Analysis of the CTL-FV formula representing the desired property is performed on a control flow graph which has been annotated with predicates derived from the program's abstract syntax tree. The set of satisfying instantiations produced by the model checker is then used to determine transformations that can be performed on the program's abstract syntax tree.

We also demonstrated the practicality of this system by presenting the implementation of three common optimizations of varying complexity using the

CTL-FV model checker. Optimizations such as dead code elimination and constant propagation translate to fairly simple CTL-FV formulas while more involved optimizations which perform code motion require more complicated formulas with specialized predicates.

Bibliography

[1] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, London, 1999.

[2] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, 2000.

[3] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *Proc. 10th Intl. Conference on Compiler Construction (CC 2001)*, April 2001.

[4] David Lacey, Niel D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2002)*, 2002.

[5] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, June 2003.

[6] Eric Van Wyk and Lijesh Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. In *Proc. Intl. Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, April 2006.